Frog

A Natural Language Processing Suite for Dutch

Version 0.13.1 - FIRST DRAFT REVISION! NOT FINAL!

Iris Hendrickx, Antal van den Bosch, Maarten van Gompel, Ko van der Sloot and Walter Daelemans
Centre for Language Studies and Centre for Speech and Language Technology, Radboud University
CLST Technical Report 16-02

June 2016

Contents

1 Introduction **1**

2 License **3**

3 Installation **5**
3.1 Manual compilation & installation 5

4 User guide **7**
4.1 Quick start guide 7
4.1.1 Interactive Mode 9
4.1.2 Server mode 9
4.2 Software usage 10
4.2.1 Character encoding 10
4.2.2 Tokenizer 10
4.2.3 Multi-word units 10
4.2.4 Lemmatizer 10
4.2.5 Morphological Analyzer 11
4.2.6 Part-of-Speech Tagger 11
4.2.7 Named Entity Recognition 11
4.2.8 Phrase Chunker 12
4.2.9 Dependency Parser 12
4.3 Using Frog from Python 12
4.3.1 Installation 13
4.3.2 Usage 13
4.4 Frog generator 14

5 Background **15**
5.1 Once upon a time 15
5.2 Module overview 15
5.2.1 Configuration file 15
5.2.2 Tokenizer 16
5.2.3 Multi-word Units 16
5.2.4 Lemmatizer 16
5.2.5 Morphological Analyzer 17
5.2.6 PoS Tagger 18
5.2.7 Named Entity Recognition 18
5.2.8 Phrase Chunker 19
5.2.9 Parser 19
5.3 Frog in practice 19

6 Credits and references **21**

A Alpino syntactic dependency labels **23**

Chapter 1

Introduction

Frog is an integration of memory-based natural language processing (NLP) modules developed for Dutch. Frog performs tokenization, part-of-speech tagging, lemmatization and morphological segmentation of word tokens. At the sentence level Frog identifies non-embedded phrase chunks in the sentence, recognizes named entities and assigns a dependency parse graph. Frog produces output in either FoLiA XML or a simple tab-delimited column format with one token per line. All NLP modules are based on Timbl, the Tilburg memory-based learning software package. Most modules were created in the 1990s at the ILK Research Group (Tilburg University, the Netherlands) and the CLiPS Research Centre (University of Antwerp, Belgium). Over the years they have been integrated into a single text processing tool, which is currently maintained and developed by the Language Machines Research Group and the Centre for Language and Speech Technology at Radboud University (Nijmegen, the Netherlands).

Chapter 2

License

Frog is free and open software. You can redistribute it and/or modify it under the terms of the GNU General Public License v3 as published by the Free Software Foundation. You should have received a copy of the GNU General Public License along with Frog. If not, see `http://www.gnu.org/licenses/gpl.html`.

Chapter 3

Installation

You can download Frog, manually compile and install it from source. However, due to the many dependencies and required technical expertise this is not an easy endeavor.

Linux users should first check whether their distribution's package manager has up-to-date packages for Frog, as this provides the easiest way of installation.

If no up-to-date package exists, we recommend to use **LaMachine**. Frog is part of our LaMachine software distribution and includes all necessary dependencies. It runs on Linux, BSD and Mac OS X. It can also run as a virtual machine under other operating systems, including Windows. LaMachine makes the installation of Frog straightforward; detailed instructions for the installation of LaMachine can be found here: `http://proycon.github.io/LaMachine/`.

3.1 Manual compilation & installation

The source code of Frog for manual installation can be obtained from Github. Because of file sizes and to cleanly separate code from data, the data and configuration files for the modules of Frog have been packaged separately.

- Source code repository: `https://github.com/LanguageMachines/frog/`
- Stable releases[1]: `https://github.com/LanguageMachines/frog/releases/`
- Frog data repository: `https://github.com/LanguageMachines/frogdata/` (required dependency!)

To compile these manually, you first need current versions of the following dependencies of our software, and compile and install them in the order specified here:

- `ticcutils`[2] - A shared utility library
- `libfolia`[3]- A library for the FoLiA format
- `ucto`[4] - A rule-based tokenizer
- `timbl`[5] - The memory-based classifier engine
- `timblserver`[6] - For server functionality around Timbl
- `mbt`[7] - The memory-based tagger

You will also need the following 3rd party dependencies:

[1] The source code repository points to the latest development version by default, which may contain experimental features. Stable releases are deliberate snapshots of the source code. It is recommended to grab the latest stable release.

[2] `https://github.com/LanguageMachines/ticcutils`

[3] `https://github.com/LanguageMachines/libfolia`

[4] `https://languagemachines.github.io/ucto`

[5] `https://languagemachines.github.io/timbl`

[6] `https://github.com/LanguageMachines/timblserver`

[7] `https://languagemachines.github.io/mbt`

- **icu** - A C++ library for Unicode and Globalization support. On Debian/Ubuntu systems, install the package `libicu-dev`.
- **libxml2** - An XML library. On Debian/Ubuntu systems install the package `libxml2-dev`.
- A sane build environment with a C++ compiler (e.g. gcc or clang), autotools, autoconf-archive, libtool, pkg-config

The actual compilation proceeds by entering the Frog directory and issuing the following commands:

```
$ bash bootstrap.sh
$ ./configure
$ make
$ sudo make install
```

To install in a non-standard location (`/usr/local/` by default), you may use the `--prefix=/desired/installation/path/` option.

Chapter 4

User guide

Frog aims to automatically enrich Dutch text with linguistic information of various forms. Frog integrates several NLP modules that perform the following tasks: tokenize text to split punctuation from word forms (including recognition of sentence boundaries and multi-word units), assignment of part-of-speech tags, lemmas, and morphological and syntactic information to words.

In the next part we first give a brief explanation on running Frog to get you started quickly, followed by a more elaborate description of using Frog and how to manipulate the settings for each of the separate modules. In Chapter 5 we discuss background information of Frog and the detailed architecture of the modules.

4.1 Quick start guide

Frog is developed as a command line tool. We assume the reader already has at least basic command line skills.

Typing `frog -h` on the command line results in a brief overview of all available command line options. Frog is typically run on an input document, which is specified using the `-t` option for plain text documents, or `-x` for documents in the FoLiA XML format. It is, however, also possible to run it interactively or as a server. We show an example of the output of Frog when processing the contents of a plain-text file `test.txt`, containing just the sentence *In '41 werd aan de stamkaart een z.g. inlegvel toegevoegd.*

We run Frog as follows: `$ frog -t test.txt`

Frog will present the output as shown in example 4.1 below:

(4.1)

1	2	3	4	5	6	7	8	9	10
1	In	in	[in]	VZ(init)	0.987660	O	B-PP	0	ROOT
2	'41	'41	['41]	TW(hoofd,vrij)	0.719498	O	B-NP	1	obj1
3	werd	worden	[word]	WW(pv,verl,ev)	0.999799	O	B-VP	0	ROOT
4	aan	aan	[aan]	VZ(init)	0.996734	O	B-PP	10	mod
5	de	de	[de]	LID(bep,stan,rest)	0.999964	O	B-NP	6	det
6	stamkaart	stamkaart	[stam][kaart]	N(soort,ev,basis,zijd,stan)	0.996536	O	I-NP	4	obj1
7	een	een	[een]	LID(onbep,stan,agr)	0.995147	O	B-NP	9	det
8	z.g.	z.g.	[z.g.]	ADJ(prenom,basis,met-e,stan)	0.500000	O	I-NP	9	mod
9	inlegvel	inlegvel	[in][leg][vel]	N(soort,ev,basis,zijd,stan)	1.000000	O	I-NP	10	obj1
10	toegevoegd	toevoegen	[toe][ge][voeg][d]	WW(vd,vrij,zonder)	0.998549	O	B-VP	3	vc
11	.	.	[.]	LET()	1.000000	O	O	10	punct

The ten TAB-delimited columns in the output of Frog contain the information we list below. This columned output is intended for quick interpretation on the terminal or in scripts. It does, however, not contain every detail available to Frog.

1. Token number (Number is reset every sentence.)

2. Token The text of the token/word

3. Lemma The lemma

4. Morphological segmentation A morphological segmentation in which each morpheme is enclosed in square brackets

5. PoS tag The Part-of-Speech tag according to the CGN tagset (**?**).

6. Confidence in the PoS tag, a number between 0 and 1, representing the probability mass assigned to the best guess tag in the tag distribution

7. Named entity type in BIO-encoding[1]

8. Base phrase chunk in BIO-encoding

9. Token number of head word in dependency graph (according to the Frog parser)

10 Dependency relation type of the word with head word

For full output, you will want to instruct Frog to output to a FoLiA XML file. This is done using the `-X` option, followed by the name of the output file.

To run Frog in this way we execute: `$ frog -t test.txt -X test.xml` The result is a file in FoLiA XML format (**?**) that contains all information in a more structured and verbose fashion. More information about this file format, including a full specification, programming libraries, and other tools, can be found on `https://proycon.github.io/folia`. We show an example of the XML structure for the token *aangesneden* in example 4.2 and explain the details of this structure in section 4.2. Each of these layers of linguistic output will be discussed in more detail in the next chapters.

(4.2)
```
<w xml:id="WP3452.p.1.s.1.w.4" class="WORD">
        <t>aangesneden</t>
        <pos class="WW(vd,vrij,zonder)" confidence="0.875" head="WW">
            <feat class="vd" subset="wvorm"/>
            <feat class="vrij" subset="positie"/>
            <feat class="zonder" subset="buiging"/>
        </pos>
        <lemma class="aansnijden"/>
        <morphology>
            <morpheme>
                <t>aan</t>
            </morpheme>
            <morpheme>
                <t>ge</t>
            </morpheme>
            <morpheme>
                <t>snijd</t>
            </morpheme>
            <morpheme>
                <t>en</t>
            </morpheme>
        </morphology>
    </w>
```

Input and Output options

By default the output of Frog is written to screen (i.e. standard output). There are two options for outputting to file (which can also be called simultaneously):

- `-o <filename>` – Writes columned (TAB delimited) data to file.
- `-X <filename>` – Writes FoLiA XML to file.

We already saw the input option `-t <filename>` for plain-text files. It is also possible to read FoLiA XML documents instead, using the `-x <filename>` option.

Besides input of a single plain text file, Frog also accepts a directory of plain text files as input `--testdir=<directory>` , which can also be written to an output directory with parameter `--outputdir=<dir>`. The FoLiA equivalent for `--outputdir` is `--xmldir`. To read multiple FoLiA documents, instead of plain-text documents, from a directory, use `-x --testdir=<directory>`.

[1] B (begin) indicates the begin of the named entity, I (inside) indicates the continuation of a named entity, and O (outside) indicates that something is not a named entity

4.1.1 Interactive Mode

Frog can be started in an interactive mode by simply typing `frog` on the command line. Frog will present a `frog>` prompt after which you can type text for processing. By default, you will press ENTER at an empty prompt before Frog will process the prior input. This allows for multiline sentences to be entered. To change this behavior, you may want to start Frog with the `-n` option instead, which tells it to assume each input line is a sentence. FoLiA input or output is not supported in interactive mode.

To exit this mode, type CTRL-D.

4.1.2 Server mode

Frog offers a server mode that launches it as a daemon to which multiple clients can connect over TCP. The server mode is started using the `-S <port>` option. Note that options like `-n` and `--skip` are valid in this mode too.

You can for example start a Frog server on port 12345 as follows: `$ frog -S 12345`.

The simple protocol clients should adhere to is as follows:

- The client sends text to process (may contain newlines)
- The client sends the string `EOT` followed by a newline
- The server responds with columned, TAB delimited output, one token per line, and an empty line between sentences.
- FoLiA input and output are also possible, using the `-x` and `-X` options without parameters. When `-X` is selected, TAB delimited output is suppressed.
- The last line of the server response consists of the string `READY`, so the client knows it received the full response.

Communicating with Frog on such a low-level may not be necessary, as there are already some libraries available to communicate with Frog for several programming languages:

- Python – **pynlpl.clients.frogclient**[2]
- R – **frogr**[3] – by Wouter van Atteveldt
- Go – **grof**[4] – by Machiel Molenaar

The following example shows how to communicate with the Frog server from Python using the Frog client in PyNLPl, which can generally be installed with a simple `pip install pynlpl`, or is already available if you use our LaMachine distribution.

```
from pynlpl.clients.frogclient import FrogClient

port = 12345
frogclient = FrogClient('localhost',port)

for data in frogclient.process("Dit is de tekst om te verwerken.")
  word, lemma, morph, pos = data[:4]
  #TODO: Further processing per word
```

Do note that Python users may prefer using the `python-frog` binding instead, which will be described in Section 4.3. This binds with Frog natively without using a client/server model and therefore has better performance.

[2] `https://github.com/proycon/pynlpl`, supports both Python 2 and Python 3
[3] `https://github.com/vanatteveldt/frogr/`
[4] `https://github.com/Machiel/gorf`

4.2 Software usage

4.2.1 Character encoding

Frog assumes the input text to be plain text in the UTF-8 character encoding. However, Frog offers the option to specify another character encoding as input with the option `-e`. This options is passed on to the *Ucto* Tokenizer. It has some limitations, (see 4.2.2) and will be ignored when the Tokenizer is disabled. The character encodings are derived from the ubiquitous unix tool *iconv* [5]. The output of Frog will always be in UTF-8 character encoding. Likewise, FoLiA XML defaults to UTF-8 as well.

4.2.2 Tokenizer

Frog uses the tokenization software *Ucto* (**?**) for sentence boundary detection and to separate punctuation from words. In general, recognizing sentence boundaries and punctuation is a simple task but recognizing names and abbreviations is essential to perfom this task well. As shown in example 4.1, the tokenizer recognizes abbreviations such as `z.g.` and `'41` and considers them to be one token. Ucto uses manually constructed rules and lists of Dutch names and abbreviations. Detailed information on Ucto can be found on `https://languagemachines.github.io/ucto/`.

The tokenizer module in Frog can be adjusted in several ways. If the input text is already split on sentence boundaries and has one sentence per line, the `-n` option can be used to prevent Frog from changing the existing sentence boundaries. When sentence boundaries were already marked with a specific marker, one can specify this marker as `--uttmarker "marker"`. The marker strings will be ignored and their positions will be taken as sentence boundaries.

If the input text is already fully tokenized, the tokenization step in Frog can be skipped altogether using the skip parameter `--skip=t`. [6]

4.2.3 Multi-word units

Frog recognizes certain special multi-word units (mwu) where a group of consecutive, related tokens is treated as one token. This behavior accommodates, and is in fact required for Frog's dependency parser as it is trained on a data set with such multi-word units. In the output the parts of the multi-word unit will be connected with an underscore. The PoS-tag, morphological analysis, named entity label and chunk label are concatenated in the same manner.

This multi-word detection can be disabled using the option `--skip=m`. When using this option, each element of the MWU is treated as separate token. We shown an example sentence in 4.2.3 that has two multi-word units: *Albert Heijn* and *'s avonds*.

```
S: Supermarkt Albert Heijn is tegenwoordig tot 's avonds laat open.

1  Supermarkt supermarkt [super][markt] N(soort,ev,basis,zijd,stan) 0.542056 O B-NP 3 su
2  Albert_Heijn Albert_Heijn [Albert]_[Heijn] SPEC(deeleigen)_SPEC(deeleigen) 1.000000 B-ORG_I-ORG B-NP_I-NP 1 ap
3  is zijn [zijn] WW(pv,tgw,ev) 0.999150 O B-VP ROOT
4  tegenwoordig tegenwoordig [tegenwoordig] ADJ(vrij,basis,zonder) 0.994033 O B-ADVP 3 predc
5  tot tot [tot] VZ(init) 0.964286 O B-PP None
6  's_avonds 's_avond ['s]_[avond][s] LID(bep,gen,evmo)_N(soort,ev,basis,gen) 0.962560 O_O O_B-ADVP 5 obj1
7  laat laat [laat] ADJ(vrij,basis,zonder) 1.000000 O B-VP 8 mod
8  open open [open] ADJ(vrij,basis,zonder) 0.983755 O B-ADJP 3 predc
9  . . [.] LET() 1.000000 O O 8 punct
```

4.2.4 Lemmatizer

The lemmatizer assigns the canonical form of a word to each word. For verbs the canonical form is the infinitive, and for nouns it is the singular form. The lemmatizer trained on the e-Lex lexicon (**?**). It is dependent on the Part-of-Speech tagger as it uses both the word form and the assigned PoS tag to disambiguate between different

[5] In the current Frog version UTF-16 is not accepted as input in Frog.

[6] In fact the tokenizer still is used, but in `PassThru` mode. This allows for conversion to FoLiA XML and sentence detection.

candidate lemmas. For example the word *zakken* used as a noun has the lemma *zak* while the verb has lemma *zakken*. Section 5.2.4 presents further details on the lemmatizer.

4.2.5 Morphological Analyzer

The morphological Analyzer (MBMA) cuts each word into its morphemes and shows the spelling changes that took place to create the word form. The fourth column in example 4.1 shows the morphemes of the example sentence. MBMA tries to decompose every token into morphemes, except for punctuation marks and names. Note that MBMA sometimes makes mistakes with unknown words such as abbreviations that are not included in the MBMA lexicon. The abbreviation `z.g.` in the example is wrongly analyzed as consisting of two parts. As shown in the earlier XML example 4.2 the past particle *aangesneden* is split into *[aan][ge][snijd][en]* where the morpheme *[snijd]* is the root form of *sned*. More information about the MBMA architecture can be found in 5.2.5.

4.2.6 Part-of-Speech Tagger

The Part-of-Speech tagger uses the tag set of *Corpus Gesproken Nederlands (CNG)* (**?**). It has 12 main PoS tags (shown in table 4.1) and detailed features for type, gender, number, case, position, degree, and tense.

We show an example of the PoS tagger output in table 4.2. The tagger also expresses how certain it was about its tag label in a confidence score between 0 (not sure) and 1 (absolutely sure). In the example the PoS tagger is very sure about the first four tokens but not about the label `N(soort,ev,basis,zijd,stan)` for the token *Psychologie* as it only has a confidence score of 0.67. *Psychologie* is an ambiguous token and can also be used as a name (tag SPEC).

ADJ	Adjective
BW	Adverb
LET	Punctuation
LID	Determiner
N	Noun
SPEC	Names and unknown
TSW	Interjection
TW	Numerator
VG	Conjunction
VNW	Pronoun
VZ	Preposition
WW	Verb

Table 4.1: The main tags in the CGN PoS-tag set.

34	Ik	VNW(pers,pron,nomin,vol,1,ev)	0.999791
35	ben	WW(pv,tgw,ev)	0.999589
36	ook	BW()	0.999979
37	professor	N(soort,ev,basis,zijd,stan)	0.997691
38	Psychologie	N(soort,ev,basis,zijd,stan)	**0.666667**

Table 4.2: The PoS tagger assigns a confidence score to each tag.

4.2.7 Named Entity Recognition

The Named Entity Recognizer (NER) detects names in the text and labels them as location (LOC), person (PER), organization (ORG), product (PRO), event (EVE) or miscellaneous (MISC).

Internally and in Frog's columned output, the tags use a so-called BIO paradigm where B stands for the beginning of the name, I signifies Inside the name, and O outside the name.

More detailed information about the NER module can be found in 5.2.7.

4.2.8 Phrase Chunker

The phrase chunker represents an intermediate step between part-of-speech tagging and full parsing as it produces a non-recursive, non-overlapping flat structure of recognized phrases in the text and classifies them with their grammatical function such as adverbial phrase (ADVP), verb phrase (VP) or noun phrase (NP). The tag labels produced by the chunker use the same type of BIO-tags (Beginning-Inside-Outside) as the named entity recognizer. We show an example sentence in 4.3 where the four-word noun phrase *het cold case team* is recognized as one phrase. The prepositional phrases (PP) consist only of the preposition themselves due to the flat structure in which the relation between prepositions and noun phrases is not expressed (note that the dependency parse labels, section 4.2.9 do express these relations). Here *Midden-Nederland* is recognized by the PoS tagger as name and therefor marked as a separate noun phrase that follows the noun phrase *de politie*.

(4.3) [Dat]$_{NP}$[bevestigt]$_{VP}$[het cold case team]$_{NP}$[van]$_{PP}$][de politie]$_{NP}$[Midden-Nederland]$_{NP}$[aan]$_{PP}$[de Telegraaf]$_{NP}$[.

1	Dat	B-NP
2	bevestigt	B-VP
3	het	B-NP
4	cold	I-NP
5	case	I-NP
6	team	I-NP
7	van	B-PP
8	de	B-NP
9	politie	I-NP
10	Midden-Nederland	B-NP
11	aan	B-PP
12	de	B-NP
13	Telegraaf	I-NP
14	.	O

Table 4.3: The phrase chunker detects phrase boundaries and labels the phrases with their grammatical information.

4.2.9 Dependency Parser

The Constraint-satisfaction inference-based dependency parser (CSI-DP) (**?**) predicts grammatical relations between pairs of tokens. In each token pair relation, one token is the head and the other is the dependent. Together these relations represent the syntactic tree of the sentence. One token, usually the main verb in he sentence, forms the root of the tree and the other tokens depend on the root in a direct or indirect relation. CSI-DP is trained on the Alpino treebank (**?**) for Dutch and uses the Alpino syntactic labels listed in appendix A. In the plain text output of Frog (example 4.1) the dependency information is presented in the last two columns. The one-but-last column shows number of the token number of the head word of the dependency relation and the last column shows the grammatical relation type. We show the last two columns of the CSI-DP output in table 4.4. The main verb *bevestigt* is root element of the sentence, the head of the subject relation (`su`) with the pronoun *Dat* and head in the object relation (`obj1`) with *team*. The noun *team* is the head in three relations: the determiner(`det`) *het* and the two modifiers(`mod`) *cold case*. The name *Midden-Nederland* is linked as an apposition to the noun *politie*. The prepositional phrase *van* is correctly assigned to the head noun *team* but the phrase *aan* is mistakenly linked to *politie* instead of the root verb *bevestigt*. Linking prepositional phrases is a hard task for parsers (**?**). More details on the architecture of the CSI-DP can be found in section 5.2.9

4.3 Using Frog from Python

It is possible to call Frog directly from Python using the `python-frog` software library. Contrary to the Frog client for Python discussed in Section 4.1.2, this library is a direct binding with code from Frog and does not use a client/server model. It therefore offers the tightest form of integration, and highest performance, possible.

1	Dat	2	su
2	bevestigt	0	ROOT
3	het	6	det
4	cold	5	mod
5	case	6	mod
6	team	2	obj1
7	van	6	mod
8	de	9	det
9	politie	7	obj1
10	Midden-Nederland	9	app
11	aan	9	mod
12	de	13	det
13	Telegraaf	11	obj1
14	.	13	punct

Table 4.4: The dependency parser labels each token with a dependency relation to its head token and assigns the grammatical relation.

4.3.1 Installation

The Python-Frog library is not included with Frog itself, but is shipped separately from `https://github.com/proycon/python-frog`.

Users who installed Frog using LaMachine, however, will already find that this software has been installed.

Other users will need to compile and install it from source. First ensure Frog itself is installed, then install the dependency `cython`[7]. Installation of Python-Frog is then done by running: `$ python setup.py install` from its directory.

4.3.2 Usage

The Python 3 example below illustrates how to parse text with Frog:

```
from frog import Frog, FrogOptions

frog = Frog(FrogOptions(parser=False))

output = frog.process_raw("Dit is een test")
print("RAW OUTPUT=",output)
output = frog.process("Dit is nog een test.")
print("PARSED OUTPUT=",output)
```

To instantiate the `Frog` class, two arguments are needed. The first is a `FrogOptions` instance that specifies the configuration options you want to pass to Frog.

The `Frog` instance offers two methods: `process_raw(text)` and `process(text)`. The former just returns a string containing the usual multiline, columned, and TAB delimiter output. The latter parses this string into a dictionary. The example output of this from the script above is shown below:

```
PARSED OUTPUT = [
 {'chunker': 'B-NP', 'index': '1', 'lemma': 'dit', 'ner': 'O',
  'pos': 'VNW(aanw,pron,stan,vol,3o,ev)', 'posprob': 0.777085, 'text': 'Dit', 'morph': '[dit]
 {'chunker': 'B-VP', 'index': '2', 'lemma': 'zijn', 'ner': 'O',
  'pos': 'WW(pv,tgw,ev)', 'posprob': 0.999966, 'text': 'is', 'morph': '[zijn]'},
 {'chunker': 'B-NP', 'index': '3', 'lemma': 'nog', 'ner': 'O',
  'pos': 'BW()', 'posprob': 0.99982, 'text': 'nog', 'morph': '[nog]'},
```

[7] Versions for Python 3 may be called `cython3` on distributions such as Debian or Ubuntu

```
 {'chunker': 'I-NP', 'index': '4', 'lemma': 'een', 'ner': 'O',
  'pos': 'LID(onbep,stan,agr)', 'posprob': 0.995781, 'text': 'een', 'morph': '[een]'},
 {'chunker': 'I-NP', 'index': '5', 'lemma': 'test', 'ner': 'O',
  'pos': 'N(soort,ev,basis,zijd,stan)', 'posprob': 0.903055, 'text': 'test', 'morph': '[test]
 {'chunker': 'O', 'index': '6', 'eos': True, 'lemma': '.', 'ner': 'O',
  'pos': 'LET()', 'posprob': 1.0, 'text': '.', 'morph': '[.]'}
]
```

There are various options you can set when creating an instance of `FrogOptions`, they are set as keyword arguments:

- `tok` – *bool* – Do tokenisation? (default: True)
- `lemma` – *bool* – Do lemmatisation? (default: True)
- `morph` – *bool* – Do morphological analysis? (default: True)
- `daringmorph` – *bool* – Do morphological analysis in new experimental style? (default: False)
- `mwu` – *bool* – Do Multi Word Unit detection? (default: True)
- `chunking` – *bool* – Do Chunking/Shallow parsing? (default: True)
- `ner` – *bool* – Do Named Entity Recognition? (default: True)
- `parser` – *bool* – Do Dependency Parsing? (default: False).
- `xmlin` – *bool* – Input is FoLiA XML (default: False)
- `xmlout` – *bool* – Output is FoLiA XML (default: False)
- `docid` – *str* – Document ID (for FoLiA)
- `numThreads` – *int* – Number of threads to use (default: unset, unlimited)

4.4 Frog generator

Frog is developed for Dutch and intended as a ready tool to feed your text and to get detailed linguistic information as output. However, it is possible to create a Frog lemmatizer and PoS-tagger for your own data set, either for a specific Dutch corpus or for a corpus in a different language. `Froggen` is a Frog-generator that expects as input a training corpus in a TAB separated column format of words, lemmas and PoS-tags.

This Dutch tweet annotated with lemma and PoS-tag information is an example of input required for `froggen`:

```
Coolio coolio TSW
mam mam N
, , LET
echt echt ADJ
vet vet ADJ
lekker lekker ADJ
' ' LET
<utt>
Eh eh TSW
... ... LET
watte wat VNW
<utt>
? ? LET
#kleinemannetjeswordengroot #kleinemannetjeswordengroot HTAG
```

Running `froggen -T taggedcorpus -c my_frog.cfg` will create a set of Frog files, including the specified configuration file, that can be used to run your own Frog. Your own instance of Frog should then be invoked using `frog -c my_frog.cfg`.

Besides the option to specify the Frog configuration file name to save the settings, one can also name the output directory where the frog files will be saved with `[-O outputdir]`. Froggen also allows an additional dictionary list of words, lemmas and PoS-tags to improve the lemmatizer module with the option `[-l lemmalist]`.

Chapter 5

Background

5.1 Once upon a time

The development of Frog's modules started in the nineties at the ILK Research Group (Tilburg University, the Netherlands) and the CLiPS Research Centre (University of Antwerp, Belgium). Most modules rely on Timbl, the Tilburg memory-based learning software package (**?**) or MBT the memory-based tagger-generator (**?**). These modules were integrated into an NLP pipeline that was first named MB-TALPA and later Tadpole (**?**). Over the years, the modules were refined and retrained on larger data sets and the latest versions of each module are discussed in this chapter. We thank all programmers who worked on Frog and its predecessors in chapter 6.

The CliPS Research Centre also developed an English counterpart of Frog, a python module called MBSP (MBSP website: `http://www.clips.ua.ac.be/pages/MBSP`).

5.2 Module overview

Frog was developed as a modular system that allows for flexibility in usage. In the current version of Frog, modules have a minimum of dependencies between each other so that the different modules can actually run in parallel processes, speeding up the performance time of Frog. The NER module, lemmatizer, phrase chunker and dependency parser are all independent from each other. All modules expect tokenized text as input. The lemmatizer, morphological analyzer and parser do depend on the PoS-tagger output. These dependencies are depicted in figure 5.1. The tokenizer and multi-word chunker are rule-based modules while all other modules are based on trained memory-based classifiers.

5.2.1 Configuration file

For advanced usage of Frog, we can define the individual settings of each module in the Frog configuration file (`frog.cfg` in the frog source directory) or adapt some of the standard options. Editing this file requires detailed knowledge about the modules and relevant options will be discussed in the next sections. You can create your own frog configuration file and run frog with `frog -c myconfigfile.cfg` . The configuation file follows the INI file format[1] and is divided in individual sections for each of the modules. Some parts of the config file are obligatory and we show the

```
-------------------------------------
[[tagger]]
set=http://ilk.uvt.nl/folia/sets/frog-mbpos-cgn
settings=morgen.settings

[[mblem]]
timblOpts=-a1 -w2 +vS
treeFile=morgen.tree

-------------------------------------
```

[1] More about the INI file format:`https://en.wikipedia.org/wiki/INI_file)`

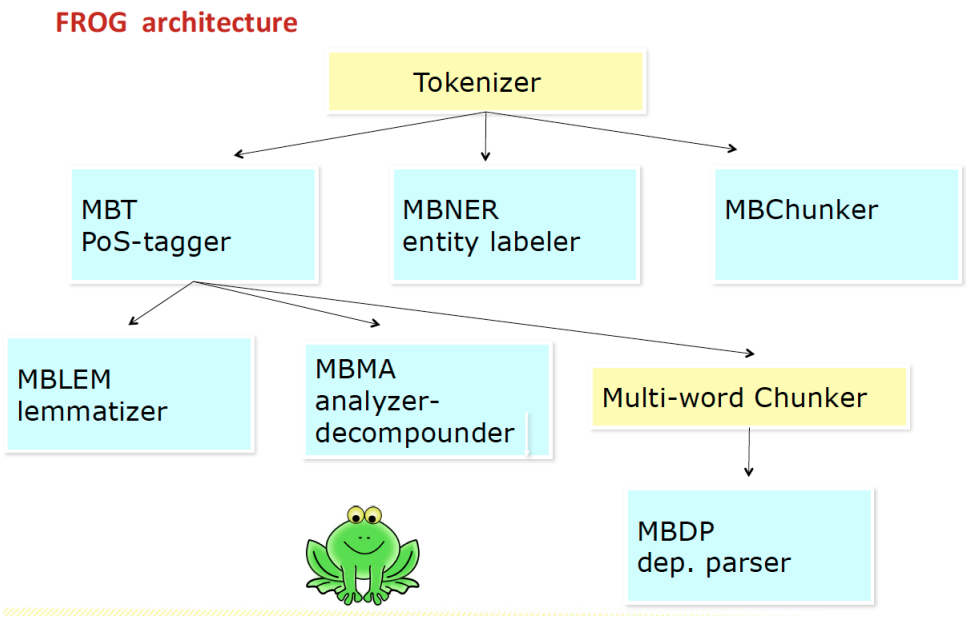

Figure 5.1: Overview of the Frog architecture and the dependencies between the modules. All blue modules are based on memory-based learning while the yellow modules are rule-based.

There are some settings that each of the modules uses:

- `debug` Alternative to using `--debug` on the command line. Debug values ranges from 1 (not very verbose) to 10??? (very verbose). Default setting is`debug=0`.
- `version` module version that will be mentioned in FoLia XML output file.
- `char_filter_file` file name of file where you can specify whether certain characters need to be replaced or ignored. For example, by default we translate all forms of exotic single quotes to the standard quote character.
- `set` reference to the appropriate Folia XML set definition that is used in the module.

5.2.2 Tokenizer

The tokenizer Ucto has its own reference guide (**?**) and more detailed information can also be found on `https://languagemachines.github.io/ucto/`.

5.2.3 Multi-word Units

Extraction of multi-word units is a necessary pre-processing step for the Frog parser. The mwu module is a simple script that takes as input the tokenized and PoS-tagged text and concatenates certain tokens like fixed expressions and names. Common mwu such as 'ad hoc' are found with a dictionary lookup, and consecutive tokens that are labeled as 'SPEC(deeleigen)' by the PoS-tagger are concatenated (`gluetag` in the Frog config file). The dictionary list of common mwu contains 1325 items and is distributed with the Frog source code and can be found under `/etc/frog/Frog.mwu.1.0` . These settings can be modified in the Frog config file.

5.2.4 Lemmatizer

The lemmatizer is trained on the e-Lex lexicon (**?**) with 595,664 unique word form - PoS-tag - lemma combinations. The e-Lex lexicon has been manually cleaned to correct several errors. A timbl classifier is trained to learn the conversion of word forms to their lemmas. Each word form in the lexicon is represented as training instance consisting

of the last 20 characters of the word form. Note that this abbreviates long words such as *consumptiemaatschappijen* to `u m p t i e m a a t s c h a p p i j e n`.In total, the training instance base has 402,734 unique word forms. As in the Dutch language morphological changes occur in the word suffix, leaving out the word beginning will not hinder the lemma assignment. The class label of each instance is the PoStag and a rewrite rule to transform the word form to the lemma form. The rewrite rules are applied to the word form endings and delete or insert one or multiple characters. For example to get the lemma *bij* for the noun *bijen* we need to delete the ending *en* to derive the lemma. We show some examples of instances in 5.1 where the rewrite rules should be read as follows. For the example the word form *haarspleten* with label `N(soort,mv,basis)+Dten+Iet` is a plural noun with lemma *haarspleet* that is derived by deleting(+D) the ending *ten* and inserting (+I) the new ending *et*. For ambiguous word forms, the class label consists of multiple rewrite rules. The first rewrite rule with the same PoS tag is selected in such case. Let's take as example the word *staken* that can be the plural noun form of *staak*, the present tense of the verb *staken* or the past tense of the verb *steken*. Here the PoS determines which rewrite rule is applied. The lemmatizer does not take into account the sentence context of the words and in those rare cases where a word form has different lemmas associated with the same PoS-tags, a random choice is made.

(5.1)

=	=	=	=	=	=	=	=	=	b	i	j	e	n	N(soort,mv,basis)+Den
=	=	=	h	a	a	r	s	p	l	e	t	e	n	N(soort,mv,basis)+Dten+Iet
=	=	=	=	h	a	a	r	s	t	u	k	j	e	N(soort,ev,dim,onz,stan)
=	=	=	=	=	=	=	=	s	t	a	k	e	n	N(soort,mv,basis)+Dken+Iak|WW(inf,nom,zonder,zonder-n)|WW(inf,prenom,zonder)|WW(inf,vrij,zonder)|WW(pv,tgw,mv)|WW(pv,verl,mv)+Daken+Ieken
=	=	=	=	=	=	=	s	p	l	e	t	e	n	N(soort,mv,basis)+Dten+Iet|WW(pv,verl,mv)+Deten+Iijten

5.2.5 Morphological Analyzer

The Morphological analyser MBMA (**?**) aims to decompose tokens into their morphemes reflecting possible spelling changes. Here we show two example words:

(5.2)
```
[leven][s][ver][zeker][ing][s][maatschappij][en]
[aan][deel][houd][er][s][vergader][ing][en]
```

Internally, MBMA not only tries to split tokens into morphemes but also aims to classify each splitting point and its relation to the adjacent morpheme. MBMA is trained on the CELEX database (**?**). Each word is represented by a set of instances that each represent one character of the word in context of 6 characters to the left and right. As example we show the 10 instances that were created for the word form *gesneden* in 5.3. The general rule in Dutch to create a part particle of a verb is to add *ge-* at the beginning and add *-en* at the end. The first character 'g' is labeled with `pv+Ige` indicating the start of an past particle (pv) where a prefix *ge* was inserted (`+Ige)`. Instance 3 represents the actual start of the verb (V) and instance 5 reflects the spelling change that transforms the root form *snijd* to the actual used form *sned* (`0+Rij>e`: replace current character 'ij' with 'e'). Instance 7 also has label 'pv' and denotes the end boundary of the root morpheme.

Timbl IGtree (**?**) trained on 3179,331 instances extracted from that were based on the CELEX lexicon of 293,570 word forms.

The morphological character type classes result in a total 2708 class labels where the most frequent class '0' occurs in 69% of the cases as most characters are inside an morpheme and to do not signify any morpheme border or spelling change. 7% of the instance represent a noun (N) starting point and 4% a verb (V) starting point. The most frequent spelling changes are the insertion of an 'e' after the morpheme (`0/e`) (klopt dit?) or a plural inflection (denoted as 'm').

The MBMA module of Frog does not analyze every token in the text, it uses the PoS tags assigned by the PoS module to filter out punctuation and names (PoS 'SPEC') and words that we labeled as ABBREVIATION by the Tokeniser. For these cases, Frog keeps the token as it is without further analysis.

Running frog with the parameter `--deep-morph` *results in a much richer morphological analysis including grammatical classes and spelling changes*.

(5.3)

1	_	_	_	_	_	_	**g**	e	s	n	e	d	e	pv+Ige
2	_	_	_	_	_	g	e	s	n	e	d	e	n	0
3	_	_	_	_	g	e	s	n	e	d	e	n	_	V
4	_	_	_	g	e	s	n	e	d	e	n	_	_	0
5	_	_	g	e	s	n	e	d	e	n	_	_	_	0+Rij>e
6	_	g	e	s	n	e	d	e	n	_	_	_	_	0
7	g	e	s	n	e	d	e	n	_	_	_	_	_	pv
8	e	s	n	e	d	e	n	_	_	_	_	_	_	0

Note that the older version of the morphological analyzer reported in (**?**) was trained on a slightly different version of the data with a context of only 5 instead of 6 characters left and right. In that older study the performance of the morphological analyzer was evaluated on a 10% held out set and an accuracy of 79% on *unseen* words was attained.

MBMA Configuration file options

When you want to set certain options for the MBMA module, place them under the heading `[[mbma]]` in the Frog configuration file.

- `set` FoLiA set name for the morphological tag set that MBMA uses.
- `clex-set` FoLiA set name PoS-tag- set that MBMA uses internally. As MBMA is trained on CELEX, it uses the CELEX POS-tag set, and not the default PoS-tag set (CGN tag set) of the Frog PoS tagger module. However, these internal pos-tags are mapped back to the CGN tag set.
- `cgn_clex_main`File name of file that contains the mapping of CGN tags to CELEX tags.
- `deep-morph` Alternative to using `--deep-morph` on the command line.
- `treeFile` Name of the trained MVMA Timbl tree (usually IG tree is used).
- `timbOpts` Timbl options that were used for creating the MBMA treeFile.

5.2.6 PoS Tagger

The PoS tagger in Frog is based on MBT, a memory-based tagger-generator and tagger[2](**?**) trained on a large Dutch corpus of 10,975,324 tokens in 933,891 sentences. This corpus is a mix of several manually annotated corpora but about 90% of the data comes from the transcribed Spoken Dutch Corpus of about nine million tokens (**?**). The other ten precent of the training data comes from the ILK corpus (46K tokens), the D-Coi corpus (330K tokens) and the Eindhoven corpus (75K tokens) citeuit den Boogaart 1975 that were re-tagged with CGN tag set. The tag set consists of 12 coarse grained PoS-tags and 280 fine-grained PoS tag labels. Note that the chosen main categories (shown in table 4.1) are well in line with a universal PoS tag set as proposed by (**?**) which has almost the same tags. The universal set has a *particles* tag for function words that signify negation, mood or tense while CGN has an *interjection* tag to label words like 'aha' and 'oké' that are typically used in spoken utterances.

5.2.7 Named Entity Recognition

The Named Entity Recognizer (NER) is an MBT classifier (**?**) trained on the SoNar 1 million word corpus labeled with manually verified NER labels. The annotation is flat and in case of nested names, the longest name is annotated. For example a phrase like 'het Gentse Stadsbestuur' is labeled as $het[Gentsestadsbestuur]_{ORG}$. 'Gentse' also refers to a location but the overarching phrase is the name of an organization and this label takes precedence. Dutch determiners are never included as part of the name. Details about the annotation of the training data can be found in the Sonar NE annotation guidelines (**?**).

The NER module does not use PoS tags but learns the relation between words and name tags directly. An easy way to adapt the NER module to a new domain is to give an additional name list to the NER module. The names on this list has the following format: the full name followed by a tab and the name label as exemplified here.The name list can be specified in the configuration file under $[[NER]]$ as `known_ners=nerfile`.

Zwarte Zwadderneel	per
LaMa groep	org

[2]MBT available at http://languagemachines.github.io/mbt/

5.2.8 Phrase Chunker

The phrase chunker module is based on the chunker developed in the 90's (**?**) and uses MBT (**?**) as classifier. The chunker adopted the BIO tags to represent chunking as a tagging task where B-tags signal the start of the chunk, I-tags inside the chunk and O-tags outside the chunk. In the context of the TTNW project (**?**), the chunker was updated and trained on a newer and larger corpus of one million words, the Lassy Small Corpus (**?**). This corpus a annotated with syntactic trees that were fist converted to a flat structure with a script.

5.2.9 Parser

The Constraint-satisfaction inference-based dependency parser (CSI-DP) (**?**) is trained on the manually verified *Lassy small* corpus(**?**) and several million tokens of automatically parsed text by the Alpino parser (**?**) from Wikipedia pages, newspaper articles, and the Eindhoven corpus.

When CSI-DP is parsing a new sentence, the parser first aims to predict low level syntactic information, such as the syntactic dependency relation between each pair of tokens in the sentence and the possible relation types a certain token can have. These low level predictions take the form of soft weighted constraints. In the second step, the parser aims to generate the final parse tree where each token has only one relation with another token using a constraint solver based on the Eisner parsing algorithm (**?**). The soft constraints determine the search space for the constraint solver to find the optimal solution.

CSI-DP applies three types of constraints: dependency constraints, modifier constraints and direction constraints. For each constraint type, a separate timbl classifier is trained. Each pair of tokens in the training set occurs with a certain set of possible dependency relations and this information is learned by the dependency constraint classifier. An instance is created for each token pair and its relation where one is the modifier and and one is head. Note that a pair always creates two instances where these roles are switched. The timbl classifier trained on this instance base will then for each token pair predict zero, one or multiple relations and these relations form the soft constraints that are the input for the general solver who selects the overall best parse tree. .The potential relation between the token pair is expressed in the following features: the words and PoS tags of each token and its left and right neighboring token, the distance between the two tokens in number of intermediate tokens, and a position feature expressing whether the token is located right or left of the potential head word.

For each token in the sentence, instances are created between the token and all other tokens in the sentence with a maximum distance of 8 tokens left and right. The maximum distance of 8 tokens covers 95% of all present dependency relations in the training set (**?**). This leads to a unbalance of instances that express an actual syntactic relation between a word pair and negative cases. Therefore, the negative instances in the training set were reduced by randomly sampling a set of negative cases that is twice as big the number of positive cases (based on experiments in (**?**)).

The second group of constraints are the modifier constraints that express for each single token the possible syntactic relations that it has in the training set. The feature set for these instances consist of the local context in 1 or 2 ?? words and PoS tags of the token.

The third group of direction constraints specify for each token in the sentence whether the potential linked head word is left or right of the word, or the root. Based on evidence in the training set, a word is added with one, two or three possible positions as soft weighted constraints. For example the token *politie* might occur in a left positioned subject relation to a root verb, a right positioned direct object relation, of in an elliptic sentence as the root form itself. We refer to chapter 5 of (**?**) for all details concerning CSI-DEP.

5.3 Frog in practice

Frog has been used in research projects mostly because of its capacity to process Dutch texts efficiently and analyze the texts sufficiently accurately. The purposes range from corpus construction to linguistic research and natural language processing and text analytics applications. We provide a overview of work reporting to use Frog, in topical clusters.

Corpus construction

Frog, named Tadpole before 2011, has been used for the automated annotation of, mostly, POS tags and lemmas of Dutch corpora. When the material of Frog was post-corrected manually, this is usually done on the basis of the probabilities produced by the POS tagger and setting a confidentiality threshold (**?**).

- The 500-million-word SoNaR corpus of written contemporary Dutch, and its 50-million word predecessor D-Coi (**?**; **?**);
- The 500-million word Lassy Large corpus (**?**) that has also been parsed automatically with the ALPINO parser (**?**);
- The 115-hour JASMIN corpus of transcribed Dutch, spoken by elderly, non-native speakers, and children (**?**);
- The 7-million word Dutch subcorpus of a multilingual parallel corpus of automotive texts (**?**);
- The *Insight Interaction* corpus of 15 20-minute transcribed multi-modal dialogues (**?**);
- The SUBTLEX-NL word frequency database was based on an automatically analyzed 44-million word corpus of Dutch subtitles of movies and television shows (**?**).

Feature generation for text filtering and Natural Language Processing

Frog's analyses can help to zoom in on particular linguistic abstractions over text, such as adjectives or particular constructions, to be used in further processing. They can also help to generate annotation layers that can act as features in further NLP processing steps. POS tags and lemmas are mostly used for these purposes. We list a number of examples across the NLP board:

- Sentence-level analysis tasks such as word sense disambiguation (**?**) and entity recognition (**?**);
- Text-level tasks such as authorship attribution (**?**), emotion detection (**?**), sentiment analysis (**?**), and readability prediction (**?**);
- Text-to-text processing tasks such as machine translation (**?**) and sub-sentential alignment for machine translation (**?**);
- Filtering Dutch texts for resource development, such as filtering adjectives for developing a subjectivity lexicon (**?**), and POS tagging to assist shallow chunking of Dutch texts for bilingual terminology extraction (**?**).

Chapter 6

Credits and references

If you use Frog for your own work, please cite this reference manual

> Frog, A Natural Language Processing Suite for Dutch, Reference guide, Iris Hendrickx, Antal van den Bosch, Maarten van Gompel en Ko van der Sloot, Language and Speech Technology Technical Report Series 16-02, Radboud University Nijmegen, Draft 0.13.1 - June 2016

The following paper describes Tadpole, the predecessor of Frog. It contains a subset of the components described in this paper:

> Van den Bosch, A., Busser, G.J., Daelemans, W., and Canisius, S. (2007). An efficient memory-based morphosyntactic tagger and parser for Dutch, In F. van Eynde, P. Dirix, I. Schuurman, and V. Vandeghinste (Eds.), Selected Papers of the 17th Computational Linguistics in the Netherlands Meeting, Leuven, Belgium, pp. 99-114

We would like to thank everybody who worked on Frog and its predecessors. Frog, formerly known as Tadpole and before that as MB-TALPA, was coded by Bertjan Busser, Ko van der Sloot, Maarten van Gompel, and Peter Berck, subsuming code by Sander Canisius (constraint satisfaction inference-based dependency parser), Antal van den Bosch (MBMA, MBLEM, tagger-lemmatizer integration), Jakub Zavrel (MBT), and Maarten van Gompel (Ucto). In the context of the CLARIN-NL infrastructure project TTNWW, Frederik Vaassen (CLiPS, Antwerp) created the base phrase chunking module, and Bart Desmet (LT3, Ghent) provided the data for the named-entity module.

Maarten van Gompel designed the FoLiA XML output format that Frog produces, and also wrote a Frog binding for Python[1], as well as a separate Frog client in Python[2]. Wouter van Atteveldt wrote a Frog client in R[3], and Machiel Molenaar wrote a Frog client for Go[4].

The development of Frog relies on earlier work and ideas from Ko van der Sloot (lead programmer of MBT and TiMBL and the TiMBL API), Walter Daelemans, Jakub Zavrel, Peter Berck, Gert Durieux, and Ton Weijters.

The development and improvement of Frog also relies on your bug reports, suggestions, and comments. Use the github issue tracker at `https://github.com/LanguageMachines/frog/issues/` or mail `lamasoftware@science.ru.nl`.

[1] `https://github.com/proycon/python-frog`
[2] Part of PyNLPL: `https://github.com/proycon/pynlpl`
[3] `https://github.com/vanatteveldt/frogr/`
[4] `https://github.com/Machiel/gorf`

Appendix A

Alpino syntactic dependency labels

This table is taken from Alpino annotation reference manual (**?**) :

dependentielabel	OMSCHRIJVING
APP	appositie, bijstelling
BODY	romp (bij complementizer))
CMP	complementizer
CNJ	lid van nevenschikking
CRD	nevenschikker (als hoofd van conjunctie)
DET	determinator
DLINK	discourse-link
DP	discourse-part
HD	hoofd
HDF	afsluitend element van circumpositie
LD	locatief of directioneel complement
ME	maat (duur, gewicht, . . .) complement
MOD	bijwoordelijke bepaling
MWP	deel van een multi-word-unit
NUCL	kernzin
OBCOMP	vergelijkingscomplement
OBJ1	direct object, lijdend voorwerp
OBJ2	secundair object (meewerkend, belanghebbend, ondervindend)
PC	voorzetselvoorwerp
POBJ1	voorlopig direct object
PREDC	predicatief complement
PREDM	bepaling van gesteldheid ‘tijdens de handeling’
RHD	hoofd van een relatieve zin
SAT	satelliet; aan- of uitloop
SE	verplicht reflexief object
SU	subject, onderwerp
SUP	voorlopig subject
SVP	scheidbaar deel van werkwoord
TAG	aanhangsel, tussenvoegsel
VC	verbaal complement
WHD	hoofd van een vraagzin